The Performance of Pascal Programs on the MULTUM

by

W. Findlay.

REPORT NO. 6.

July 1974.

1. Introduction.

When the Multum computer was ordered in July 1972 the manufacturer offered only a macro-assembler to assist in programming. This was clearly inadequate for software developments on the scale then envisaged, so it was decided to implement Pascal [1], using the 1900 series compiler [2] as a basis. The project team was formed in October. Analysis of the 1900 compiler and design work for the Multum compiler continued during the winter months, but it was not until April 1973, and the availability of a large 1900 computer with the GEORGE 3 system, that significant progress could be made.

A viable compiler was finally achieved by the end of August 1973. For historical reasons, this compiler is called G Pascal B.

In this report G Pascal B object programs are analysed, using a set of short test routines made available by Wirth [4] (and reproduced as the Appendix). It is hoped that the analysis will be of use:

(a) in giving Pascal programmers a "feel" for the characteristics of the Multum implementation;

(b) in "tuning" Pascal programs for minimum time or space; and

(c) in guiding future improvements to the compiler.

For each Pascal construct we present figures showing the size of the object code, the number of instructions obeyed, and the execution time. As an interesting comparison, similar figures are given for XPAC 1B, the 1900 series compiler of Welsh and Quinn [2]. Note that times are given in units of 100 nanoseconds, to aid comparison with Wirth's figures for the CDC 6400 computer (see the Appendix).

2. The Multum ALP/2 Computer.

The Multum ALP/2 is a "midi" computer, with a word length of 16 bits. All instructions are 1 word long. There are facilities in the instruction set for manipulating single bits, 8-bit bytes, words and doublewords. The addressing unit is the word, but several of the addressing modes scale index quantities. That is, an index is multiplied by 2 or divided by 2 before the addition of the base, according as the operand is a double word or a byte. Addresses and literal operands are restricted to 8 bits by the 16-bit instruction format. Consequently, the directly-addressable storage is limited to 256 words relative to some base. The instruction address register (S) can be used as such a base in the jump instruction, and in accessing constants held with the program code; the data pointer register (P) gives access to 256 words of read/write storage. G Pascal B uses the P register to base the activation record of each procedure, allowing/

allowing direct addressing of simple variables and indirect access to structured variables.

The clock rate of the machine is 8MHz and the store cycle time is 650 nanoseconds, but the virtual memory system and the store interface logic show this down to an effective access time of 1 microsecond (10 units). Apart from degenerate cases the shortest instruction execution time is 1.375 microseconds (13.75 units) for a fetch with literal operand. The average instruction time is about 2 microseconds (20 units).

3. The G Pascal B Compiler.

G Pascal B was implemented by means of a "half bootstrap" using the 1900 series XPAC compiler as the basis. First, the code generation and table handling routines were rewritten to produce macro-calls in the Multum assembly language, Usercode. Compiling this version on the 1900 resulted in a 1900-Multum cross-compiler. When made to compile itself this created a compiler expressed in Usercode and capable of generating Usercode. Finally, an assembly of this text yielded G Pascal B.

The compiler accepts a large subset of (unrevised) Pascal [1], the following being the main restrictions:

(a) file declarations are not allowed;

(b) powerset declarations are not allowed;

(c) real and alfa are not allowed;

(d) non-local goto is not allowed;

(e) procedures and functions cannot be used as formal parameters; and

(f) the symbol packed is not allowed.

Two extensions were made:

(a) the value part was implemented, allowing global variables to be initialised at compile time; and

(b) the linkto statement was added to allow the use of external Usercode routines.

The compiler occupies about 52K words: 36K of code and 16K of data. It compiles large programs at a rate of about 720 lines per minute, corresponding to 120 words of object code per second. (These figures are very approximate - there is no accurate way of measuring CPU time on the Multum).

The size of the data area allows for the compilation of much larger/

larger programs than the compiler itself. About 1½K of the code is due to the run-time library and about 2½K is used in subscript checking (a compile time option). This leaves about 32K words of code, to be compared with about 16K words for XPAC. The difference is due to several factors:

(a) the use of in-line code instead of subroutines in many contexts (perhaps 5%);

(b) inefficiencies in the code generator (10%);

(c) complexities resulting from the generation of Usercode rather than machine binary (20%);

(d) problems with the instruction set, especially jumps (25%); and

(e) difficulties with the addressing modes, especially in accesses to structures (40%).

These points are brought out more fully in the subsequent analysis.

With hindsight, it is clear that the Multum is oriented to the human programmer, in ways a compiler cannot easily exploit. (This remark applies to all high-level languages, not just to Pascal.)

4. Analysis.

An analysis of the results is presented in this section; the figures themselves are given in section 5, following. A number of points should be kept in mind.

(a) The figures for object code size and for the number of instructions obeyed were obtained by compiling the test programs and examining the code manually. They are thus exact, except when the number of instructions obeyed may vary at run time. If there are just two cases both figures are given, but where there are three cases or more an unweighted average is given instead.

(b) The Multum execution times were obtained by summing the execution times for the generated instructions, as given in the manufacturer's specifications, and then rounding to the nearest multiple of 100 nanoseconds. Since there is no store interleaving or pipelining this gives a valid figure for the execution time of an instruction sequence. Multiple cases have been dealt with in the way outlined above (a).

(c) When a feature is not implemented its figures are replaced by "NI"; an unknown value is denoted by "?".

The numbering of the following subsections parallels that of section 5, for ease of cross-reference.

4.1 /

4.1 T1: integer expressions.

Integer expressions are compiled simply and efficiently by both compilers, but there are four points worth mentioning.

The 1900 has a "store zero" (STOZ) order which XPAC exploits to compile "i:=0" into one instruction. Similarly the 1900 "negated fetch" (NGX) order allows "k:=-i" to be compiled into two instructions. Neither of these optimisations is possible on the Multum.

The Multum code for "k:=i*j" is:

```
SETA i      / fetch i to A
MLTA j      / product to (A,B)
LDRA B      / contract product to A
STAS k      / store A in k
```

In this special case the code could be improved by omitting the third instruction and storing B instead of A. (The 1900 code contains two instructions to contract the product).

The Multum fixed-point division order (DIVE) is peculiarly unsuited to Pascal: it works on 31-bit (!) fractional operands, requiring a preliminary shift; and it truncates quotients towards zero, requiring adjustment if the result is negative. Contrasting with this, the 1900 (DVS) order does exactly what is required.

4.2 T2: Boolean expressions

These examples expose a weakness of the G Pascal B compiler: its handling of forward jumps.

Both XPAC and G Pascal B treat a Boolean expression <B> by compiling code suggested by the conditional expression:

if <B> then true else false

The operators and, or are treated similarly:

(a) <B1> and <B2> as

if <B1> then <B2> else false

(b) <B1> or <B2> as

if <B1> then true else <B2>

Such transformations increase efficiency when an explicit Boolean value is not needed (the normal case, in conditionals and loops), with no loss of efficiency when it is needed (as in these examples). However, they do introduce a plethora of forward jumps.

Conditional/

Conditional jumps are coded on the Multum as a combination of two orders, the conditional skip and the general unconditional jump. The latter has two forms, a relative jump of short range ($\pm$ 128 words) and an indirect jump; the indirect form must be used when the range of the jump is unknown. Due to the strictly one-pass nature of the compiler this applies to all forward jumps. Four words of code are generated by the combination of a skip with an indirect jump. The effect of this can be seen in code sizes for the Boolean operators, which use the construction twice, and in the relational operators, which use it once. The problem might be tackled in two ways:

(a) by pooling the indirect addresses in literal tables (this would reduce a jump to three words);

(b) by determining the maximum range of a forward jump and using the short form where possible (this would reduce some jumps to just two words).

Neither approach has been feasible with the present interface between the assembler and the compiler. It is noteworthy that this defect of the Multum instruction set is troublesome even to the (human) assembly-language coder.

There are no such problems on the 1900 which has a series of orders combining a test and a jump in a single instruction.

4.3 T3: character expressions

No comment.

4.4 T4: real expressions

Not implemented on the Multum.

4.5 T5: optimisation of integer arithmetic

Neither compiler produces special code for literal operands which are powers of 2. The improved times for the Multum are due to the constant operands (no data fetch). The increased code for division on the 1900 is caused by the lack of a "divide by literal" order.

4.6 T6: control structures

The full force of the remarks in 4.2, above, is brought home by a comparison of the Multum and 1900 coding for control structures: especially as the figures given for the Multum represent a "best case". In particular, the coding of loops can be increased by 2 words and the execution time can be increased by 10n when the controlled statement occupies more than 128 words (the limit of a relative backward jump).

G Pascal B performs some optimisations of simple for statements, such as those presented here. The simplest "general" form of these statements would occupy 15 words, obeying 5+4n instructions in a time of 100+88n units. This is still superior (in terms of overhead per repetition) to the 1900 coding. In fact, XPAC makes no attempt to optimise for statements. It certainly should do: the present coding uses two jump instructions per repetition, significantly impeding execution on the "pipelined" 1906A and 1906S processors. (As a result the examples shown are no faster on a 1906A than on the MULTUM!)

4.7 T7: arrays

Due to the short address-part of a Multum instruction it is not generally possible to exploit the fact that arrays in Pascal have a fixed size. Instead an "array word" containing a base address is set up for each array, on entering the block in which it is decleared. Unfortunately, the Multum does not have an addressing mode which will add the contents of an array word and a (scaled) index register. This mode must be simulated by fetching the array word into the accumulator (A register) and using a mode with the accumulator as the base register. The extra fetch accounts for 1 of the 4 instructions required to fetch "a[i]". Worse still - this method cannot be used when storing into "a[i]" from the A register. To handle this the address of "a[i]" must be pre-calculated and saved in a "temporary" for later use as an indirect address, incurring a penalty of 2 or more instructions per access. The problem has been compounded by a software design error. A further 1 instruction per access could be saved by holding the address of the (notional) "zeroth" element in the array word, rather than the address of the "first" element.

It should be noted that all character arrays are packed by default in the Multum implementation. The intention was two-fold: to save space and to compensate for the loss of standard type "alfa". Nonetheless, this too was a design error. The resultant increase in the complexity of the compiler significantly increased debugging time. And despite this complexity the object code for accessing packed arrays is even less efficient than that for unpacked arrays, thus annihilating any saving in data storage. Packing should certainly be implemented, but after the bootstrap, and in the manner of the revised language. This allows the programmer to use it only when the saving outweighs the cost. (Of course, we expected that for character arrays the saving would outweigh the cost but we were not then sufficiently familiar with the dismal properties of the Multum indexing hardware.)

4.8 T8: records

Records are treated by both compilers as if they were arrays, the field selectors representing constant subscripts. Thus access to records shares the same features as access to arrays, see 4.7 above.

The assignment operations on complete records deserve mention. In-line code is used on the Multum for the assignment of arrays and records, as there is no convenient way to pass parameters to a sub-routine (within the present framework). The 1900 also uses in-line code, but is able to exploit a "store-to-store copy" (MOVE) instruction.

There are upwards of 20 structured assignments in the G Pascal B compiler, accounting for about 1½% of its size!

Note that in T8 and in T7, real quantities in Wirth's original programs/

programs have been replaced by integer quantities. This makes the Multum and 1900 figures directly comparable (real is not implemented on the Multum). As reals occupy 2 words on the Multum and the 1900, but only 1 word on the CDC6400 the comparison of these machines with the 6400 is not entirely invalid, despite the alteration.

4.9 T9: sets

Not implemented on the Multum.

4.10 T10: procedures and functions

The Multum and the 1900 are remarkably similar in the work they perform to implement procedures, as measured by the number of instructions obeyed.

On both machines non-local variables (other than globals) are accessed by means of a "static chain". This is relatively inefficient on the Multum because the single base (P) register must be saved and restored at each access. On the other hand variable (var) parameters are handled less efficiently on the 1900, which lacks indirect addressing. Global variables are treated specially: the 1900 uses direct addressing and the Multum exploits "memory-held registers". A memory-held register is one of 8 words of main store, addressable relative to the contents of P. One of the addressing modes permits an operand to be located by adding the contents of a memory-held register to a small displacement held in the instruction, The only penalties are an extra store cycle (10 units) per access, and an overhead on procedure entry (to set up the global base addresses).

The differences in size for procedure calls show the use of out-of-line routines by the 1900. This would not be profitable on the Multum.

4.11 T11: dynamic variables and pointers

The Multum implements "alloc(p)" ("new(p)" in the CDC6400 version) by in-line code, the 1900 calls a subroutine.

The figures for access to structures based on pointers are partly a reflection of the remarks made in 4.7, above. However there is a residual inefficiency in the Multum coding which could be eliminated by a simple "peephole" optimisation. This would reduce the figures 8 and 12 to 6 and 8 words respectively. (A similar inefficiency is present in the 1900 coding, which could be improved from 5 and 7 to 4 and 6 words in the same cases.)

5. The/

5. The figures

5.1 T1: integer expression

Pascal statement	Multum code words	Multum code obeyed	time	1900 code words	1900 code obeyed
i := 0	2	2	36	1	1
i := 10	2	2	36	2	2
k := i	2	2	46	2	2
k :=-i	3	3	60	2	2
k := i+j	3	3	70	3	3
k := i*j	4	4	115	5	4/5
k := i div j	6	4/5	140	3	3
k := i mod j	5	4/5	148	3	3
k := abs(i)	4	3/4	61/76	4	3/4
k := sqr(i)	4	4	115	4	4
k := trunc(x)	NI	NI	NI	3	3
k := int(c)	2	2	46	2	2

5.2 T2: Boolean expressions

Pascal statement	Multum words	Multum obeyed	time	1900 words	1900 obeyed
p := true	2	2	36	2	2
p := q	2	2	46	2	2
p := not q	3	3	60	3	3
p := p or q	14	7	126	8	5
p := p and q	14	7	126	8	5
p := i<j	10	6/7	126/123	6	5/6
p := i<=j	10	6/7	126/123	6	5/6
p := i=j	10	6/7	126/123	6	5/6
p := p<q	10	6/7	126/123	6	5/6
p := p<=q	10	6/7	126/123	6	5/6
p := p=q	10	6/7	126/123	6	5/6

5.3 T3: character expressions

Pascal statement	Multum words	Multum obeyed	Multum time	1900 words	1900 obeyed
c := 'D'	2	2	36	2	2
c := d	2	2	46	2	2
c := chr(i)	2	2	46	2	2

5.4 T4: real expressions
Not implemented on the Multum.

5.5 T5: optimisation of integer arithmetic

Pascal statement	Multum words	Multum obeyed	Multum time	1900 words	1900 obeyed
i := 2*j	4	4	96	5	4/5
i := 10*j	4	4	98	5	4/5
i := j div 2	6	4/5	130	4	4
i := n div 2	6	4/5	130	4	4
m := n mod 8	5	4/5	128	4	4

5.6 T6: control structures

Pascal statement	Multum words	Multum obeyed	Multum time	1900 words	1900 obeyed
if p then	5	3	63/53*	2	2
if i=j then	6	4	86/76*	3	3
if i<j then	6	4	86/76*	3	3
if p then else	7	3/4*	63/76*	3	2/3*
in the following "n" is the number of repetitions					
while p do	6	4n + 3	66n + 63	3	3n - 1
repeat until p	3	3n - 1	53n - 14	2	2n
for i := 1 to 10 do	7	3n + 3	65n + 59	10	7n + 6
for i := 10 downto 1 do	7	3n + 3	65n + 59	10	7n + 6
case i of					
1:; .... ; m :	8 + 3m	8	164	2 + 2m	4
end					

*condition false/condition true

5.7 T7: arrays

Pascal statement	Multum words*	Multum obeyed*	Multum time	1900 words*	1900 obeyed*
x := a[5]	4(6)	4(6)	85	2	2
x := a[i]	5(7)	5(7)	109	3	3
x := a[i+1]	8(10)	8(10)	169	5(6)	5(6)
x := c[5,10]	4(6)	4(6)	85	2	2
x := c[i,j]	9(11)	9(11)	248	5	5
n := c[i+1,j-1]	15(17)	15(17)	346	10(11)	10(11)

5.8 T8: records

Pascal statement	Multum words*	Multum obeyed*	Multum time	1900 words	1900 obeyed
x := d.x	4(6)	4(6)	85	2	2
x := d.z.im	4(6)	.4(6)	85	2	2
x := d.y[i]	4(6)	4(6)	95	3	3
x := u[i].x	6(8)	6(8)	178	4	4
x := u[i].y[j]	9(11)	9(11)	233	5	5
z := d.z	23	25	515	3	3
d := u[i]	25	68	1920	5	5

5.9 T9: sets
Not implemented on the Multum.

*the figures in parenthesis apply to an assignment in the opposite direction.

5.10 T10: procedures and functions

Pascal statement	Multum words	Multum obeyed	Multum time	1900 words	1900 obeyed
in p3:					
a := b	5	5	111	3	3
a := c	2	2	56	2	2
in p2:					
a := b	2	2	56	2	2
p3(a,b,x)	11	20	459	9	21
in f					
f := 1	2	2	36	2	2
in q1:					
x := 1	2	2	46	3	3
a := x	2	2	66	3	3
in q2:					
r	NI	NI	NI	2	5+?
in the program part:					
p0	5	14	303	1	11
p1(a)	7	16	359	4	22
p2(a,a)	9	18	415	6	24
q1(a)	8	17	363	4	22
q2(p0)	NI	NI	NI	7	26
a := f(a)	8	17	431	7	24

5.11 T11: dynamic variables and pointers

Pascal statement	Multum words	Multum obeyed	Multum time	1900 words	1900 obeyed
alloc(p)	10	8	144/164*	4	13/10*
q := nil	3	3	50	2	2
q := p	2	2	46	2	2
q := p↑.l	4	4	85	3	3
q := p↑.r↑.l	8	8	170	5	5
q := p↑.l↑.r↑.l	12	12	255	7	7

* result = nil/result ≠ nil

6. Conclusion

The implementation of Pascal on the Multum has been moderately successful. Simple constructions are very efficient, but some features of the language are handled less well. Difficulties arise from two sources:

(a) deficiences of the Multum, in particular flaws in the addressing scheme; and

(b) the one-pass nature of the compiler.

These criticisms should be seen in context. It is hardly surprising that the structure of a compiler which originated on computers with "big machine" architecture turns out to be less than ideal on a "small machine". And in defence of the Multum, it appears that the size of the object code on comparable computers would be even larger (perhaps by as much as a quarter).

The problems can be diminished by extensive special-case coding. To some extent this is done and is reflected in the size of the compiler. However, the most effective improvements (e.g. in handling data structures) would require the gathering of comprehensive statistics about the particular program being compiled. This is feasible only if the single-pass approach is abandoned. There are indications that a good two-pass compiler could achieve savings of 25% in the size of the object code. Such a compiler would be about the same size, in bits, as XPAC.

ACKNOWLEDGEMENTS

Thanks are due to Professor Wirth for his advice and encouragement, to Dr. Welsh for the use of XPAC and much practical assistance, and to the group who undertook the labour of performing the bootstrap: J. Cavouras, R. Cupples and P. Morton. J. Davis provided an improved version of the library routines and I. Christie wrote a special-purpose assembler for the object macros.

REFERENCES

[1] N. Wirth, "The Programming Language Pascal", Acta Informatica, Vol 1, No. 1, 1971; pp 35-63.

[2] J. Welsh and C. Quinn, "A Pascal Compiler for the ICL1900 Series Computers", Software - Practice and Experience, Vol 2, No. 1, 1972; pp 73-77.

[3] N. Wirth, "The Design of a Pascal Compiler", Software - Practice and Experience, Vol 1, No. 4, 1971; pp 309-333.

[4] N. Wirth, Private Communication, 1973.

APPENDIX The test programs

We reproduce here the texts of the 12 tests defined by Wirth, along with his figures for the execution times on the CDC 6400 computer. Like the times given for the Multum these are in units of 100 nanoseconds (one clock pulse on the CDC machine), and were obtained by summing the execution times of the separate instructions. However Wirth cautions that the actual performance will be better than these figures indicate, because of store interleaving and optimisations which may reduce the number of store accesses.

Test T12 is not mentioned in the body of this report: it uses packed structures and could not be implemented on either the 1900 or the Multum.

```
        {T1-T5: STANDARD AND SUBRANGE TYPE EXPRESSIONS}
   VAR I,J,K: INTEGER;
       M,N; 0..9999;
       P,Q: BOOLEAN;
       C,D: CHAR;
       X,Y,Z: REAL;
BEGIN {T1: INTEGER EXPRESSIONS}
   I := 0;                                                    15
   J := 10;                                                   16
   K := I;                                                    27
   K := -I;                                                   33
   K := I+J;                                                  40
   K := I*J;                                                 102
   K := I DIV J;                                             136
   K := I MOD J;                                             188
   K := ABS(I);                                               38
   K := SQR(I);                                               79
   K := TRUNC(X);                                             48
   K := ORD(C);                                               27
{T2: BOOLEAN EXPRESSIONS}
   P := TRUE;                                                 16
   P := Q;                                                    27
   P := ¬Q;                                                   31
   P := P∨Q;                                                  39
   P := P∧Q;                                                  39
   P := I<J;                                                  57
   P := I≤J;                                                  57
   P := I=J;                                                  58
   P := P<Q;                                                  39
   P := P≤Q;                                                  50
   P := P=Q;                                                  50
{T3: CHARACTER EXPRESSIONS}
   C := 'D';                                                  16
   C := D;                                                    27
   C := CHR(I);                                               27
{T4: REAL EXPRESSIONS}
   X := 1.0;                                                  27
   X := Y;                                                    27
   X := -Y;                                                   33
   X := I;                                                    40
   Z := X+Y;                                                  52
   Z := X*Y;                                                  91
   Z := X/Y;                                                  91
   Z := ABS(X);                                               38
   Z := SQR(X);                                               79
{T5: OPTIMIZATION OF INTEGER ARITHMETIC}
   I := 2*J;                                                  33
   I := 10*J;                                                 45
   I := J DIV 2;                                              39
   I := N DIV 2;                                              33
   M := N MOD 8;                                              45
END .
```

```
        { T6: CONTROL STRUCTURES }
    VAR P: BOOLEAN; I,J: INTEGER;
BEGIN   {N = NO. OF REPETITIONS}
    IF P THEN ;                          25/17
    IF I = J THEN ;                      43/35
    IF I < J THEN ;                      43/35
    IF P THEN ELSE ;                     25/30
    WHILE P DO ;                         25+30n
    REPEAT UNTIL P ;                     25n-8
    FOR I := 1 TO 10 DO ;                36+64n
    FOR I := 10 DOWNTO 1 DO ;            36+64n
    CASE I OF                            43
   1: ;   2: ;   3:
    END ;
END .

        {T7-T9: STATIC DATA STRUCTURES }
    TYPE INDEX = 1..20;
       ROW = ARRAY [INDEX] OF REAL;
       COMPLEX = RECORD RE, IM: REAL END ;
       ITEM = RECORD X: REAL; Y: ROW;  Z: COMPLEX END ;
    VAR X: REAL; Z: COMPLEX;
       I,J: INDEX; P: BOOLEAN;
       D: ITEM;
       A,B: ROW;
       C: ARRAY [INDEX] OF ROW;
       U: ARRAY [INDEX] OF ITEM;
       R,S,T: SET OF INDEX;
BEGIN {T7: ARRAYS}
    X := A[5];                           27
    X := A[I];                           39
    X := A[I+1];                         39
    X := C[5,10];                        27
    X := C[I,J];                         74
    X := C[I+1, J-1];                    74
{T8: RECORDS AND ARRAYS}
    X := D.X;                            27
    X := D.Z.IM;                         27
    X := D.Y[I];                         39
    X := U[I].X;                        102
    X := U[I].Y[J];                     121
    Z := D.Z;                            54
    D := U[I];                          1127
{T9: SETS}
    R := [5]                             16
    S := [2,3,5,7,11,13,17,19];          27
    T := [I];                            39
    T := [I,J];                          73
    R := S V T;                          39
    R := S Λ T;                          39
    R := S - T;                          39
    IF I IN S THEN ;                    54/46
    P := I IN S;                         68
END .
```

```
        {T10: PROCEDURES AND FUNCTIONS}
    VAR A: INTEGER;
    PROCEDURE PO;  BEGIN END ;
    PROCEDURE P1(X: INTEGER);  BEGIN END ;
    PROCEDURE P2(X,Y: INTEGER);
       VAR B: INTEGER;
       PROCEDURE P3(X,Y,Z: INTEGER);
            VAR C: INTEGER;
          BEGIN
             A := B                    39
             A := C;                   27
          END ;
       BEGIN {P2}
          A := B;                      27
          P3(A,B,X);                  189
       END ;

    FUNCTION F(X: INTEGER) : INTEGER;
       BEGIN
          F := 1;                      16
       END ;
    PROCEDURE Q1(VAR X: INTEGER);
       BEGIN
          X := 1;                      28
          A := X;
       END ;

    PROCEDURE Q2(PROCEDURE R);
       BEGIN
          R                            48
       END ;

BEGIN
    PO;                               102
    P1(A);                            129
    P2(A,A);                          156
    Q1(A);                            118
    Q2(PO);                           118
    A := F(A);                        156
END .

        {T11: DYNAMIC VARIABLES AND POINTERS}
    TYPE REF = ↑ NODECLASS;
       NODE = RECORD L,R: REF END ;
    VAR P,Q: REF;
       NODECLASS: CLASS OF NODE;
    BEGIN NEW(P);                     68
       Q := NIL;                      16
       Q := P;                        27
       Q := P↑.L;                     39
       Q := P↑.R↑.L;                  51
       Q := P↑.L↑.R↑.L;               63
    END .
```

```
        {T12: PACKED RECORDS}
VAR I: INTEGER; P: BOOLEAN; C: CHAR;
   R,S: PACKED RECORD
              A: -10..+10;
              B: BOOLEAN;
              C: CHAR;
              D: 0..255;
              E: -1..+1;
              F: 1..10;
        END ;
BEGIN
   I := R.A;                          33
   P := R.B;                          44
   C := R.C;                          44
   I := R.D;                          44
   I := R.E;                          39
   I := R.F;                          38
   R.A := I;                          72
   R.B := P;                          72
   R.C := C;                          72
   R.D := I;                          72
   R.E := I;                          72
   R.F := I;                          66
END .
```